Implementierung - Transportband „Fabrik der Zukunft“

Scherlin, Erik

Braun, Florian

Großmann, Moritz

10. Juli 2017

Inhaltsverzeichnis

1 Einführung

Der Transportmanager bildet das Bindeglied zwischen der Steuerung und dem Transportband der Anlage. Neben der Weiterleitung der Befehle muss die Software Schlitten koordinieren und fertiggestellte Aufträge zurückmelden. Die besondere Herausforderung bei diesem Projekt war allerdings, mehrere Schlitten gleichzeitig transportieren zu können. Auf diese Anforderung ist das gesamte Konzept und das Softwaredesign ausgelegt. Der eventorientierte Ansatz erleichtert es, Anwendungen mit vielen nebenläufigen Aufgaben zu schreiben. Die Objekte kommunizieren so direkt untereinander und bekommen nur Nachrichten, die für sie auch wirklich relevant sind. Außerdem ist die Verbindung der Objekte untereinander über Pointer effizienter als eine Speicherung und Verwaltung in einem großen Baustein. Um Softwarebausteine besser auf einzelne Gruppenmitglieder verteilen zu können wurde eine hohe Modularität angestrebt. Diese wird durch die Bereitstellung von Interfaces erreicht, die die Funktionen definieren, die ein bestimmtes Modul benötigt. Zudem ist so eine Entwicklung eines Moduls möglich, ohne dass ein anderes benötigtes Modul fertiggestellt ist. Das Testen wird ebenfalls erleichtert, da man wenig Abhängigkeiten hat, die man ebenfalls testen müsste. Da die Anlage ganz unter dem Motto Industrie 4.0 steht, sollte die Software auch deren Anforderungen umsetzen. Eine Erweiterung oder ein Umbau der Anlage wurde von vornherein beim Design berücksichtigt und stellt kein Problem dar. Auch eine Aufrüstung mit Sensoren, Stoppern und Stationen ist über ein Konfigurationsfile möglich.

2 Applikationsgliederung

2.1 Packages

Um Applikationen übersichtlich zu halten, werden in Java Packages verwendet. Diese verhalten sich ähnlich wie Namespaces in anderen Programmiersprachen. Die Packages sind dabei in zwei große Gruppen unterteilt. Packages, die mit *transportsystem* beginnen, enthalten Interfaces und Klassen, die unabhängig von der Implementierung die Anlage repräsentieren. Typen in diesen Packages sollten auch dringlichst unabhängig von konkreten Implementierungen sein, da sie sonst nicht mehr in anderen Applikationen verwendet werden könnten. Im Folgenden werden die einzelnen Packages erklärt.

transportsystem.machinedesc In diesem Package sind Interfaces, die für das Parsen der XML-Anlagenbeschreibung benötigt werden. Sie definieren die benötigten Setter-Methoden, mit denen Daten aus der Beschreibung gesetzt werden können. Zusätzlich ist hier die Klasse *XMLParser* und das Interface *IModelFactory* abgelegt. Letzteres wird von *XMLParser* benötigt, um die Elemente aus dem XML-File in konkrete Objekte zu übersetzen, die das passende Interface implementieren. Eine Übersicht über alle Klassen und Interfaces ist in Abbildung 1 dargestellt.

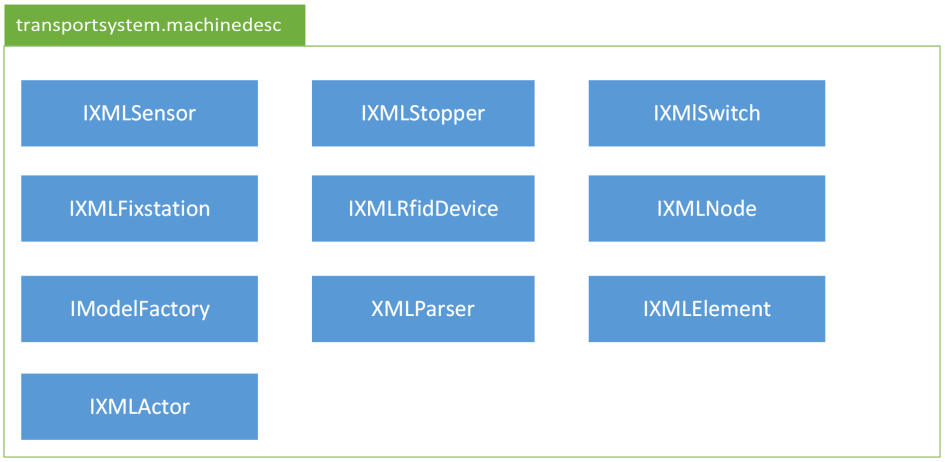

Abbildung 1: Übersicht über die Klassen im Package *transportsystem.machinedesc*

transportsystem.models.components In diesem Package sind Interfaces, die die Funktionen der einzelnen Elemente auf der Bahn beschreiben. Zu den Elementen gehören die physischen Bestandteile der Anlage, also Sensoren, Stopper, Weichen, etc. Zusätzlich sind noch Typen vorhanden, die von den Elementen benötigt werden. Eine Übersicht aller Interfaces ist in Abbildung 2 dargestellt.

transportsystem.models.communication Interfaces in diesem Package definieren, welche Funktionalität ein Element auf der Bahn benötigt, um mit der Anlage zu kommunizieren. Um auch hier wieder eine Modularisierung zu erreichen, gibt es für jeden Elementtyp ein Interface. Es wäre so also möglich, Weichen auf eine andere Art als Stopper oder Sensoren anzusprechen. Dieses Vorgehen erlaubt es wieder, die Kommunikationslogik zu ändern, ohne dass andere Komponenten davon betroffen sind. Eine Übersicht aller Interfaces ist in Abbildung 3 dargestellt.

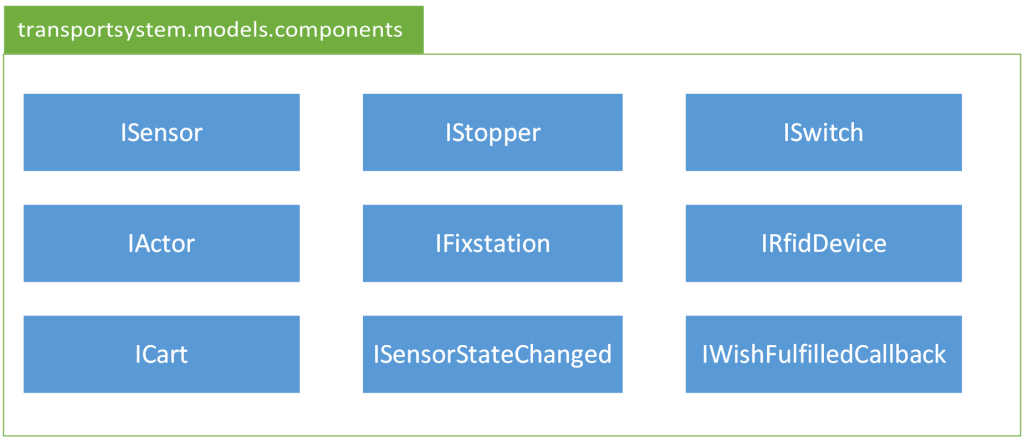

Abbildung 2: Übersicht über die Klassen im Package *transportsystem.models.components*

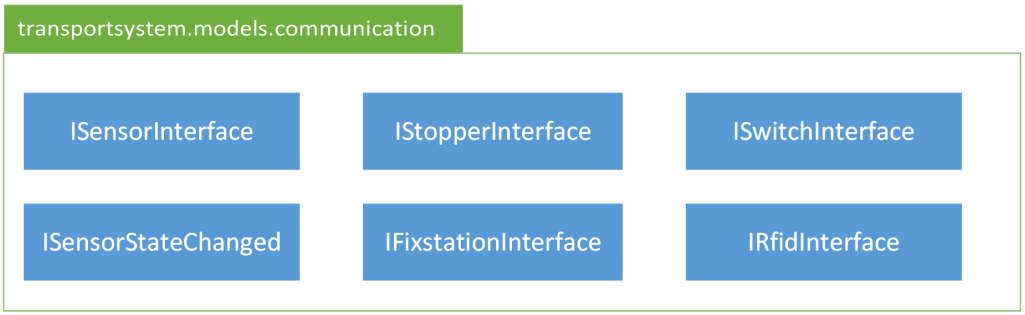

Abbildung 3: Übersicht über die Klassen im Package *transportsystem.models.communication*

Die zweite große Gruppe bilden die Packages, die mit fdzss17 beginnen. Sie können teilweise oder komplett gegen eine andere Implementierung ausgetauscht werden. In ihnen befinden sich alle Typen, die speziell die Implementierung des Sommersemesters 17 betreffen und nicht allgemeingültig sind.

fdzss17 Hauptpackage der Entwicklungen im Sommersemester 2017. Dort befinden sich hochgradig spezielle Entwicklungen wie z. B. die Implementierung der *ModelFactory* für den XML-Parser oder die Klasse, die die Startargumente richtig liest und prüft.

fdzss17.communication In diesem Package sind alle Klassen untergebracht, die mit den angrenzenden Systemen Steuerung, SPS und der Rfid-SPS kommunizieren. Zudem finden sich hier auch noch Klassen, die das Kommuni-

kationsprotokoll abhandeln. Eine Übersicht aller Klassen ist in Abbildung 4 dargestellt.

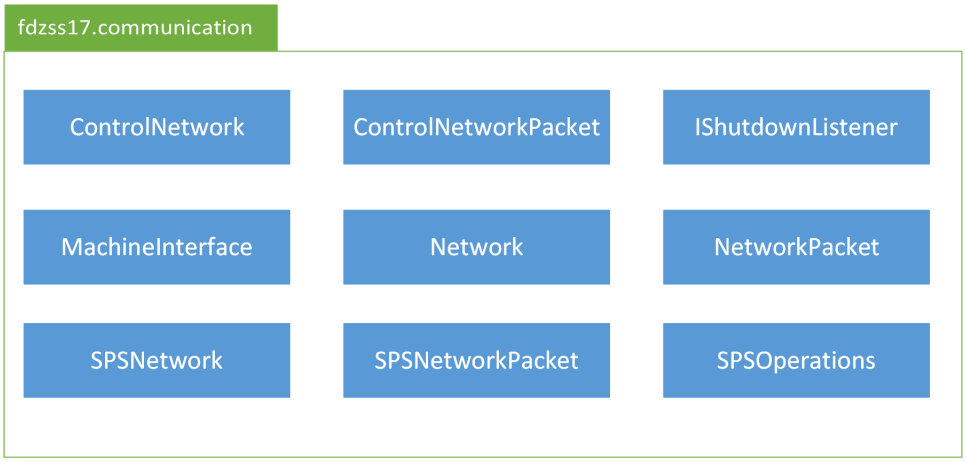

Abbildung 4: Übersicht über die Klassen im Package *fdzss17.communication*

fdzss17.components Die konkreten Implementierungen der Interfaces aus transportsystem.models.components finden sich hier wieder. Die Klassen beschreiben die genaue Funktionsweise der einzelnen Elemente auf der Bahn und die Kommunikation untereinander. Eine Besonderheit stellt die Klasse *Node* dar, sie implementiert kein gesondertes Interface, welches die Funktionalität beschreibt, da sie dafür zu speziell ist. Außerdem ist ein Node nicht physisch auf der Anlage vorhanden. Eine Übersicht aller Klassen ist in Abbildung 5 dargestellt.

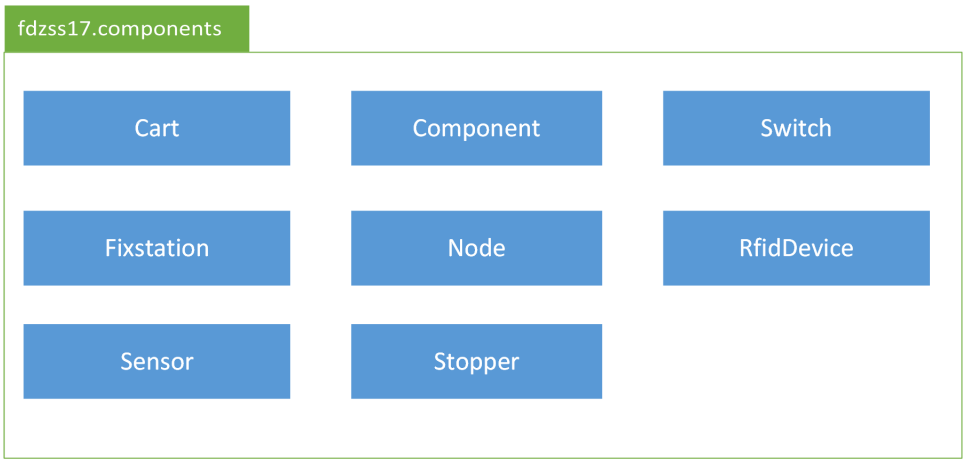

Abbildung 5: Übersicht über die Klassen im Package *fdzss17.components*

fdzss17.ordermanagement Dieses Package dient zur Gruppierung mehrerer Klassen, die sich um die Auftragsverwaltung innerhalb der Anwendung kümmern. Die Verwaltung ist so speziell, dass es hierzu kein Pendant in den *transportsystem.**-Packages gibt. Eine Übersicht aller Klassen ist in Abbildung 6 dargestellt.

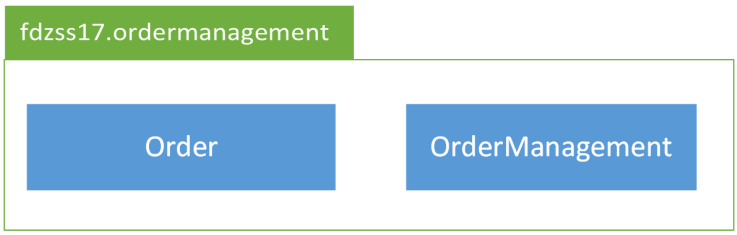

Abbildung 6: Übersicht über die Klassen im Package *fdzss17.ordermanagement*

2.2 Komponenten

Um die Abhängigkeiten im Programm so gering wie möglich zu halten, wurden Funktionalitäten zu Modulen zusammengefasst, die gegeneinander austauschbar sind. Dafür bietet jedes Modul Schnittstellen an, die von anderen Modulen implementiert werden, damit diese kompatibel sind.

Kommunikationsschichten Abstrahiert die Kommunikation zu angrenzenden Systemen. Empfängt Befehle einer übergeordneten Einheit, interpretiert diese und leitet sie an die entsprechenden Module zur Weiterverarbeitung weiter. Stellt ebenfalls ein Verbindungsstück zum Ausführen von Operationen auf der Anlage dar.

XML-Parser Unabhängige Komponente, die anhand der Beschreibungsdatei Instanzen von Klassen erstellt, die die Elemente im XML-File repräsentieren. Dabei ist die Beschreibungsdatei und die Parserkomponente unabhängig von der Implementierung der Instanzen.

Auftragsmanagement Verwaltungskomponente, die im Transportmanager die ausstehenden Aufträge verwaltet und anderen Komponenten zur Verfügung stellt. Zudem übernimmt sie das korrekte Anlegen von Aufträgen aus den Daten, die die Steuerung sendet.

Bahnmanagement Verwaltet die Positionen der einzelnen Schlitten auf der Anlage. Diese werden in Queues gespeichert und beinhalten Informationen, um auch an einem Knoten ohne angeschlossenes Rfid-Gerät Schlitten mit Palette behandeln zu können.

Anlagenkomponenten Die einzelnen Teile, die physisch auf der Bahn vorhanden sind, sind auch als Softwarebausteine implementiert. Diese besitzen Attribute und Funktionen, die eine Steuerung ermöglichen. Diese Bausteine werden unter dem Oberbegriff Anlagenkomponenten zusammengefasst.

3 Komponentenbeschreibung

3.1 Kommunikationsschicht

Die Kommunikationsschicht setzt sich aus zwei verschiedenen Teilen zusammen, dem Kommunikationsteil mit dem Controller sowie den SPS Steuerungen. Für die Datenübertragung wurden insgesamt drei Socketverbindungen eingesetz. Zwei zu den Steuerungen und eine zum Controller. Hierfür steht eine Oberklasse *Network* zur Verfügung, von der sich zwei Klassen für beide Kommunikationsrichtungen ableiten (siehe Abb. 7). Diese abstrakte Klasse implementiert das Netzwerkhandling. Hierfür stehen die Methoden connect(), reconnect() sowie disconnect() zur Verfügung, welche die Socketverbindung für alle abgeleiteten Klassen übernimmt. Anschließend wird über die Methode *read()* bzw. *readPacket()* ein Paket über den Socket gelesen und kann anschließend verarbeitet werden.

Für das abarbeiten der einzelnen Netzwerkpakete werden die Methoden *readPacketHeader()*, sowie *readPacketPayload()* definiert, welche spezialisiert von jeder Unterklasse implementiert werden müssen. Für die Verwaltung der einzelnen Pakete ist nach demselben Muster die Methode *processPacket()* zum Abarbeiten der zuvor gelesenen Pakete vordefiniert. Des weiteren ist eine Methode für das setzen der Paketlänge sowie eine zum Senden von Paketen definiert, welche ebenfalls überschrieben werden muss.

Die einzelnen Netzwerkpakete sind in der abstrakten Klasse *NetworkPacket* gekapselt. Hierin enthalten sind die Daten der einzelnen Netzwerkpakete, welche über Methoden greifbar sind.

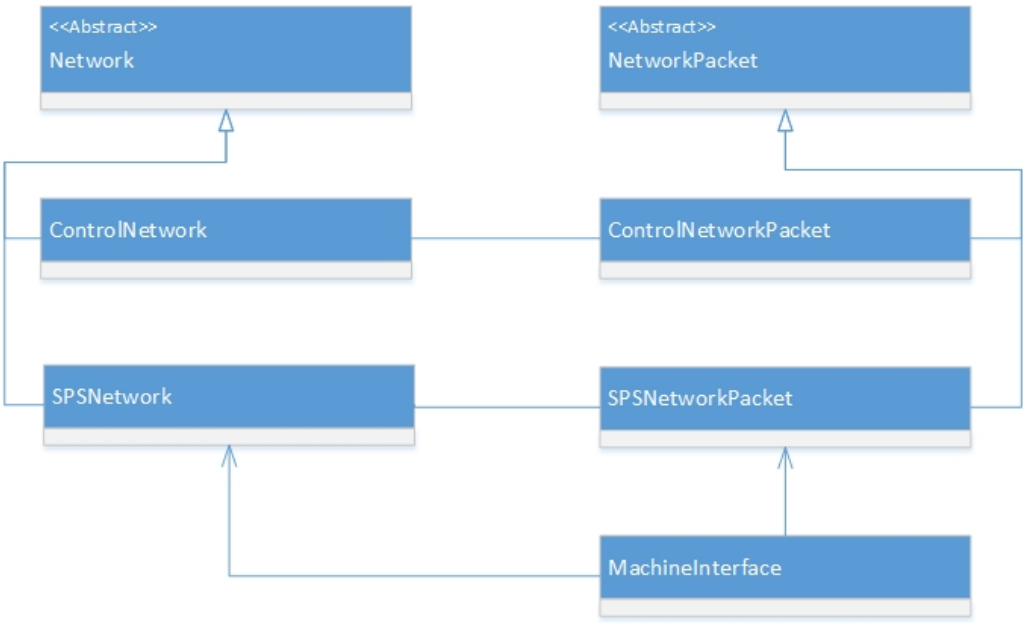

Abbildung 7: Klassendiagramm - Übersicht Netzwerkkommunikation

3.1.1 Kommunikationsschicht Controller

Die Schnittstelle für die Kommunikation mit dem Control stellt die Klasse *ControlNetwork*, welche sich, wie in Abbildung 8 beschrieben, von der abstrakten Klasse Network ableitet.

Die von *NetworkPacket* abgeleitete Klasse *ControlNetworkPacket* bündelt die Datenpakete für diese Schicht. Hierbei werden alle im Protokoll definierten Instanzvariablen in der Klassenstruktur abgebildet und über getter-Methoden nach außen bereitgestellt. Die Klasse bietet zwei Konstruktoren mit verschieden Argumenten an, wobei einer zum Lesen und der andere zum Senden eines Control-Netzwerkpakets dient.

Im Konstruktor der Klasse wird ein Thread aus dem ThreadPool angefordert, welcher Zyklisch zum einen eingehende Nachrichten liest und zum anderen die Liste mit ausgehenden Nachrichten abarbeitet. Die gelesenen werden mithilfe der Methode processPacket() verarbeitet. Hierbei wird die Nachricht in ihre einzelnen Teile zerlegt und je nach angefragtem Typ ein neuer Auftrag der Klasse Order erzeugt. Dies geschieht mithilfe des Interfaces *IOrderManagement*, welches von der Klasse OrderManagement implementiert wird. Nach dem erfolgreichen anlegen des Auftrags wird das erste Acknowlegde zurückgesendet. Dem erzeugten Auftrag wird eine Instanz von *IOrderFinishedCallback* mitgegeben, welcher bei dessen Abschluss aufgerufen wird. Das *ControlNetwork* implementiert wiederum das Interface und

wird so durch den Callback aufgerufen, sobald der zweite Acknowledge für einen Auftrag gesendet werden muss. Für den Fall, dass des „Shutdown“-Kommandos wird eine Methode zum registrieren von listenern bereitgestellt. Hierzu muss die interessierte Klasse das Interface *IShutdownListener* implementieren und sich auf die Instanz registrieren. Sobald das Kommando empfangen wird, werden alle registrierten Listener benachrichtigt und können den geregelten „Shutdown“-Prozess durchführen.

Die *ControlNetwork* Klasse stellt ebenfalls die Funktionalität für das persistente Speichern des Anlagenzustands. Dieser kann sowohl nach senden des offiziellen „Shutdown“-Befehls als auch nach Programmabsturz ausgeführt werden. Hierfür werden die Listen mit allen noch nicht abgeschlossenen Aufträgen sowie die Liste mit allen noch zu sendenden Acknowledge-Nachrichten abgespeichert. Das Speichern erfolgt nach jeder Änderung in einer der beiden Listen, hierfür werden nur die empfangenen bzw. zu sendenden Netzwerkpakete gespeichert. Nach erneutem Programmstart wird abgefragt, ob - falls vorhanden - eine gesicherte Datei geladen oder verworfen werden soll. Nach dem einlesen der Sicherungsdatei wird der Programmablauf nachsimuliert. Dazu werden alle serialisierten Empfangspakete, welche alle offenen Aufträge enthalten, einfach an die Empfangslogik übergeben und die normale Logik erledigt den Rest. Die zu sendenden Pakete werden in den Sendepuffer kopiert und wie gehabt versendet, sobald die Ressourcen zur Verfügung stehen.

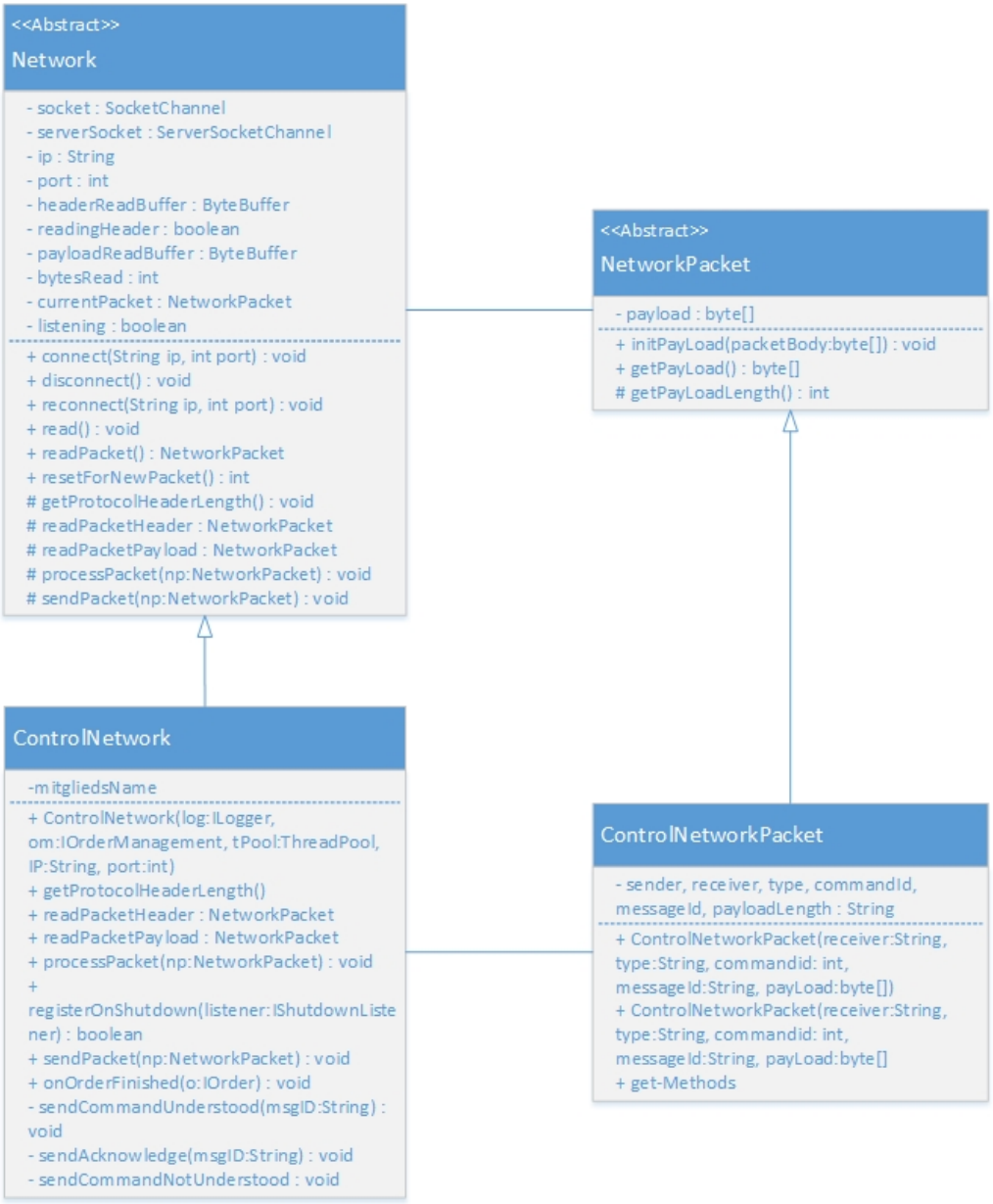

Abbildung 8: Klassendiagramm - Kommunikation zur Steuerung

3.1.2 Kommunikationsschicht SPS

Die Kommunikationsschicht zur SPS wurde von der Verbindung äquivalent zum Controller aufgebaut. Das heißt, dass beide sich die selbe Oberklasse teilen und die selben Schritte für das Verbindungshandling durchführen. Die komplette Logik wird durch ein MachineInterface realisiert. Dies stellt alle Funktionalitäten zum Steuern der Aktoren sowie lesen der Sensoren. So wird die Kapselung der Netzwerkkomponente am besten gewährleistet. Der komplette Aufbau sowie die Integration in das Systemumfeld ist in Siehe

Abbildung 7.

Folgende Befehle können vom Transportmanager über das MachineInterface an die Steuerung gesendet werden:

1. setStopper - zum setzen der Stopper über die Enumeration Stopper-Position
2. setFixstation - für das fixieren/loslassen der Fixierstationen
3. setPosition - zum schalten einer Weiche
4. requestRFID - anfrage zum lesen eines RFID-Lesers
5. writeRFID - beschreiben eines RFID-Tags

Die Schnittstelle ist als Singleton designed, das bedeutet, dass die einmalig erstellte Instanz mittels *initialize()* instantiiert wird und über *getInstance()* geholt wird. Bei der Initialisierung wird der Klasse die Informationen der Netzwerkverbindungen mitgegeben, da diese über die Programmstartparameter konfiguriert und von der Programminitialisierung verwaltet werden. Dadurch bietet sich der Vorteil, dass konkurrierende Zugriffe besser bearbeitet werden können, da jeder auf die selbe Instanz zugreift. Ebenfalls sind dort die einmalig instantiierten Netzwerkverbindungen verfügbar. Die Flusskontrolle der Netzwerkpakete ist mittels synchronized Blöcken realisiert. Da eine sequenzielle Sende- und Empfangslogik verwendet wird, ist dies ohne Probleme möglich. Nach jedem Befehl oder jeder Anfrage wird zuerst auf die Antwort der Nachricht gewartet, bevor die nächste versendet wird, um die Flusskontrolle und ein direktes abarbeiten zu gewährleisten.

Alle Anfragen werden über die Methode *spsCommandResponse()* als Netzwerkpaket der Klasse SPSNetworkPacket versendet. Jedes einzelne Paket hält einen Byte-Stream inne, welcher alle relevanten Daten, wie die Kommandonummer und gebündelt den Payload in sich trägt. Der Payload bildet die Bitcodierte Anfrage an die SPS. Darin enthalten der Befehlstyp, welcher in SPSOperations als Enumeration abgebildet ist und vom Netzwerk beim Empfangen eines Pakets wieder aufgelöst werden kann. Ebenfalls im Payload enthalten, die ID, des zu setzenden Aktors, sowie der Zustand, welcher gesetzt werden soll. Beim Empfangen wird das Netzwerkpaket wieder zu einem SPSNetworkPacket gemappt. Der Rückgabewert, ob die Operation erfolgreich war, wird im Payload mit übergeben.

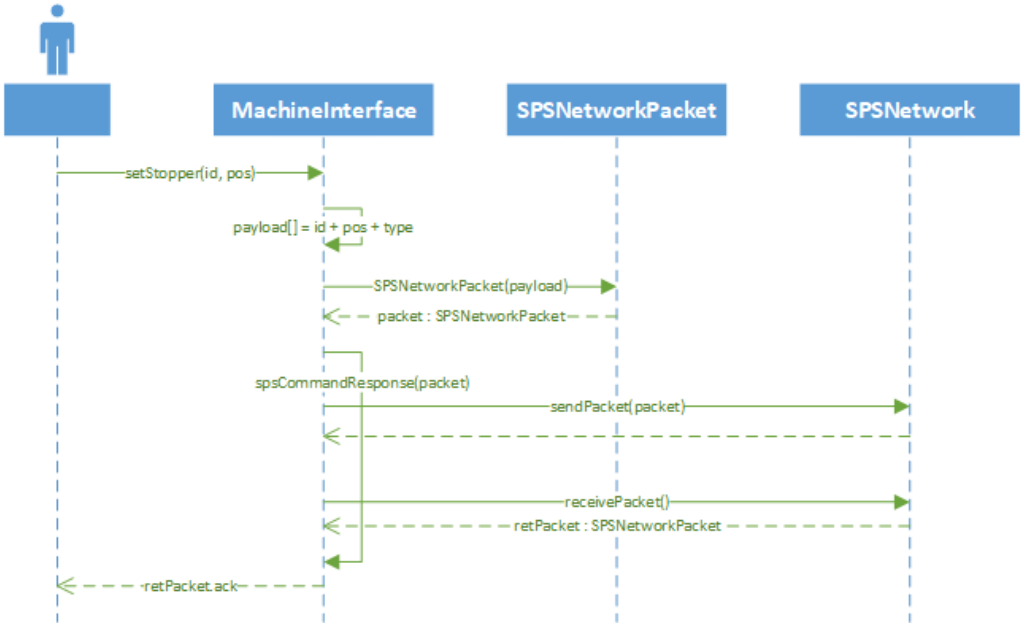

Abbildung 9: Sequenzdiagramm - Anfrage an die Steuerung

Das lesen und schreiben der RFID-Tags erfolgt über eine gesonderte Netzwerkverbindung, da diese durch eine separate Steuerung repräsentiert wird. Für die Anfrage zum lesen eines Tags steht die Methode *requestRFID()* zur Verfügung, welche auf die synchronisierte Methode *readRFID()* zugreift. Diese stellt eine Anfrage zum lesen über die übergebene Leser-ID. Sollte die ID nicht verfügbar sein oder nicht existieren wird kein gültiger Wert zurückgesendet. War das lesen erfolgreich, so wird die gelesene ID des Chips ganzzahlig zurückgeliefert. Für das beschreiben der Tags, was im normalen Programmablauf nicht vorgesehen ist, wurde die Methode *writeRFID* geschrieben, welche Tags über ein Lesegerät neu beschreibt.

Für das Bereitstellen der Sensordaten ist ein eigener Thread verantwortlich, welcher zyklisch über die Methode *updateStates()* die Daten von der SPS anfordert. Dieser wird über die Methode *startListeningForSensorStates()* gestartet. Für das benachrichtigen der interessierten Modells steht die Methode *registerListener()* zur Verfügung. Dadurch kann sich jeder Interessent, welcher das Interface *ISensorStateChange* implementiert, auf einen Sensor registrieren. Sollte nun ein Sensor seinen Zustand ändern, werden alle registrierten Teilnehmer benachrichtigt und können darauf reagieren. Damit keine Informationen des aktuellen Betriebs während des Datenaustauschs verloren gehen, wurde in die SPS eine Logik zum zwischenspeichern der einzelnen Sensorwerte eingebaut. Hierbei wird sich der aktuell gelesene Wert solange gemerkt, bis er von der Steuerung ausgelesen wurde.

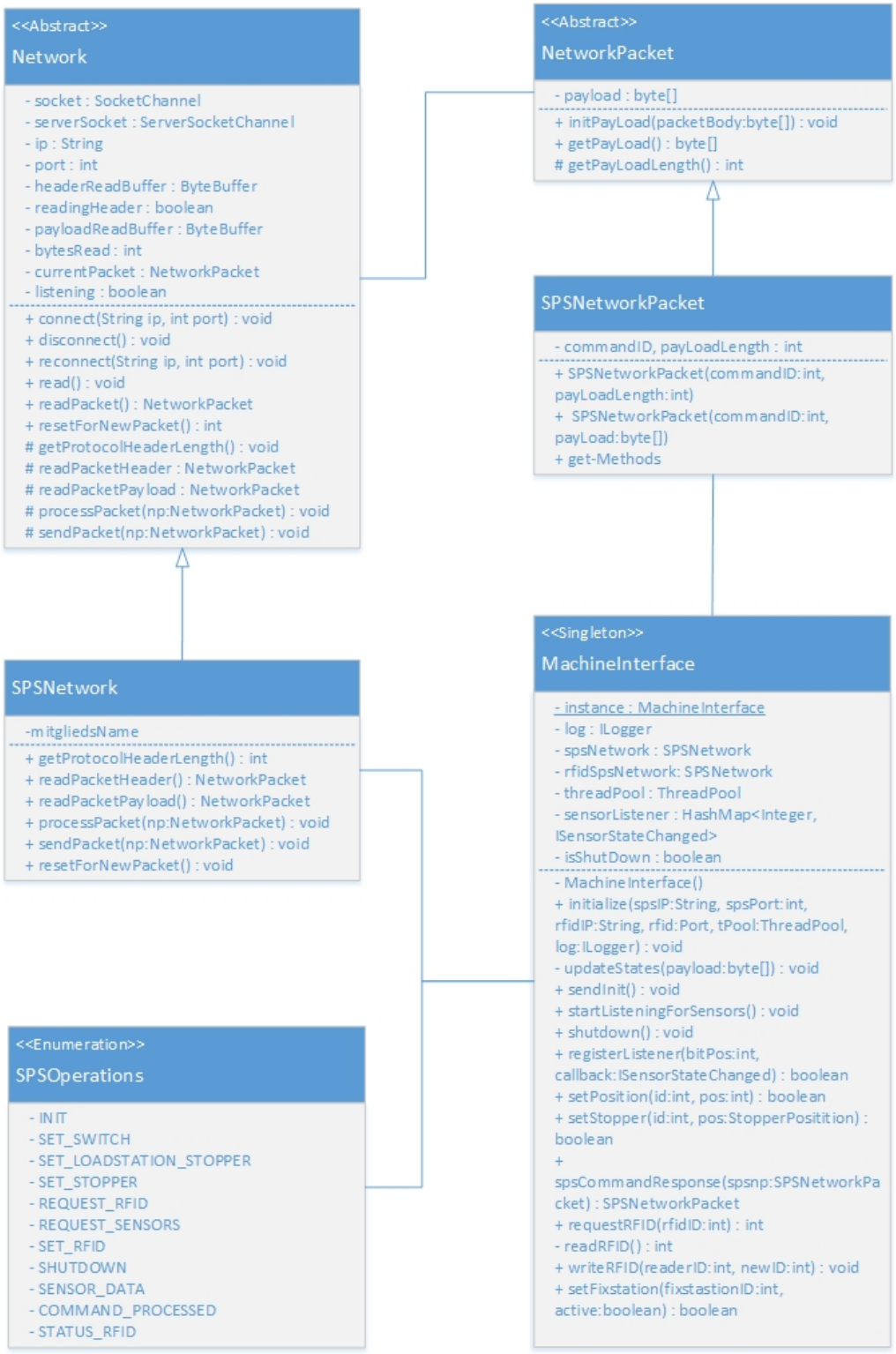

Abbildung 10: Klassendiagramm - Kommunikation zur Steuerung

3.2 Parser für Anlagenbeschreibung

Die Parserkomponente ist unabhängig vom Rest des Programms verwendbar. Sie kann alle XML-Files einlesen, die eine gültige Anlagenbeschreibung mit den Komponenten eines Transportsystems enthalten, da diese nur eine abstrakte Beschreibung sind. Die konkrete Implementierung dieser Komponenten ist anwendungsspezifisch.

Die einzelnen Komponenten und ihre Attribute sind durch Interfaces definiert, die das Präfix „IXML“ haben und ebenfalls im Package *transportsystem.machinedesc* sind. Das Interface *IModelFactory* definiert Methoden, mit denen Objekte erzeugt werden können, deren Klasse das jeweilige Komponenteninterface implementiert. In Abbildung 12 ist dargestellt, wie der XML-Parser initialisiert und eine Datei eingelesen werden kann.

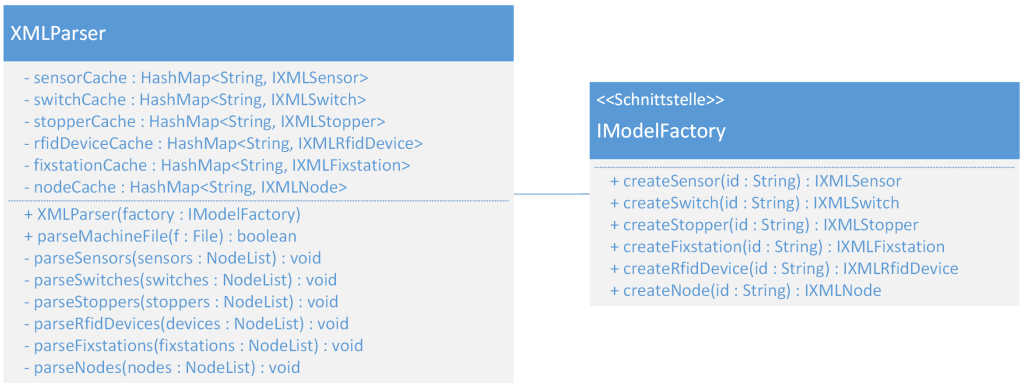

Abbildung 11: Zusammenhang zwischen XMLParser und IModelFactory

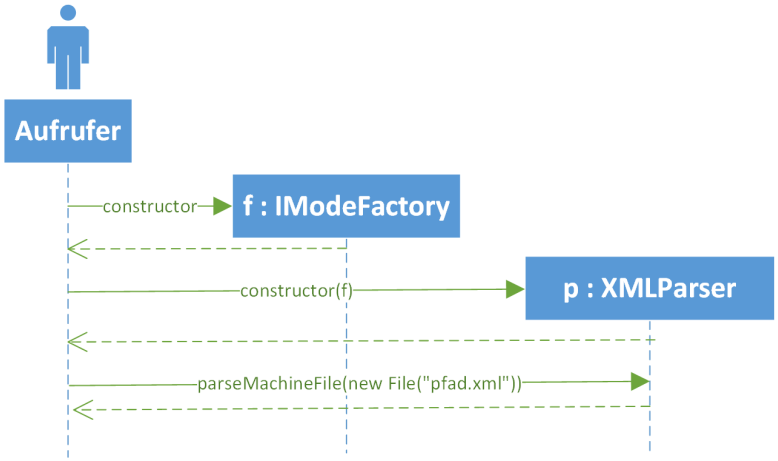

Abbildung 12: Grober Ablauf wie die Anlagenbeschreibung eingelesen wird

Der Parser arbeitet die Datei nicht von oben nach unten ab sondern

beachtet die nachfolgende Reihenfolge.

1. Sensoren
2. Weiche
3. Stopper
4. RFID-Geräte
5. Fixierstationen
6. Knoten

Die Reihenfolge ist absichtlich so gewählt, da Abhängigkeiten so aufgelöst werden können. Komponenten werden nach ihrem Erzeugen und Initialisieren im Parser zwischengespeichert, damit diese zum Initialisieren von anderen Elementen verwendet werden können. Alle Elemente in der XML-Datei besitzen deswegen eine eindeutige Identifikation, welche auch zur Referenzierung innerhalb der Datei dient.

Die Sensoren werden als erstes eingelesen, sie benötigen nur eine Angabe, an welcher Position sich das Bit befindet, das den Zustand des Sensors darstellt. Wie die Bits angeordnet sind ist in Kapitel 3.5 im Abschnitt „Sensoren“ beschrieben. Der genaue Aufbau der Setter ist in Abbildung 13 dargestellt. Eine Beispieldefinition für Sensoren ist in Abbildung 14 dargestellt.

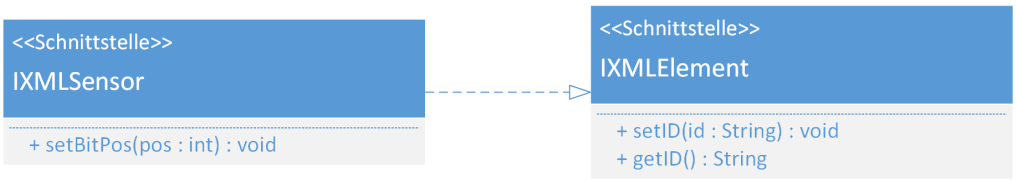

Abbildung 13: Übersicht über die Setter-Methoden eines Sensors für XML-Attribute

```
<sensor id="B3.3" bitPosition="7" />
<sensor id="B9" bitPosition="6" />
<sensor id="B9.1" bitPosition="5" />
<sensor id="B14" bitPosition="4" />
```

Abbildung 14: Beispieldefinition von Sensoren in der Beschreibungsdatei

Die Weichen enthalten als Informationen eine zusätzliche Identifikation, welche im Attribut „spsId“ gespeichert wird. Diese gibt an, mit welchem Wert die Weiche in der SPS angesprochen werden kann. Zusätzlich enthält sie verschiedene Kindelemente. Mit diesen können Ein- und Ausfahrtssensoren definiert werden, die der Weiche signalisieren, wann ein Schlitten die Weiche passiert hat. Zusätzlich können Bahnen definiert werden, die die Weiche anfahren kann. Zum Bahnnamen selbst wird auch der Wert, die „value“ gespeichert, mit der die Weiche von der SPS in diese Position gefahren werden kann. Der genaue Aufbau der Setter ist in Abbildung 15 dargestellt. Eine Beispieldefinition von Weichen ist in der Abbildung 16 dargestellt.

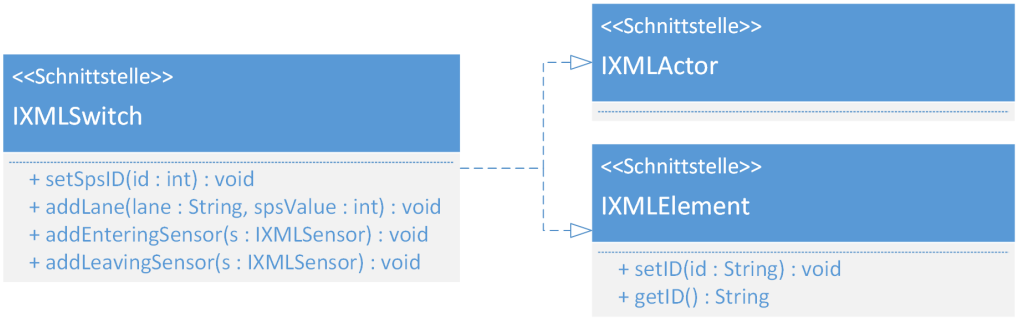

Abbildung 15: Übersicht über die Setter-Methoden einer Weiche für XML-Attribute

```
<switch spsId="1" id="ea-switch">
    <enteringSensor id="B3.4" />
    <enteringSensor id="B5.1" />

    <leavingSensor id="B4" />
    <leavingSensor id="B3.3" />

    <position lane="mainlane" value="1" />
    <position lane="ealane" value="0" />
</switch>
```

Abbildung 16: Beispieldefinition von Weichen in der Beschreibungsdatei

Stopper enthalten ebenfalls eine „spsId“ mit der sie vom Programm angesprochen werden können. Außerdem kann jeweils genau ein Ein- und Ausfahrtssensor definiert werden. Um eine Stauvermeidung zu ermöglichen kann ein Vorgängerstopper eingetragen werden, auf den dann bei Bedarf zugegriffen werden kann. Der genaue Aufbau der Setter ist in Abbildung 17 dargestellt. Eine Beispieldefinition von Weichen ist in der Abbildung 18 dargestellt.

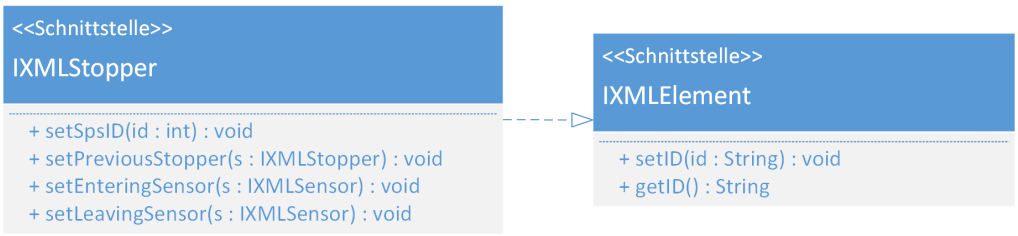

Abbildung 17: Übersicht über die Setter-Methoden eines Stoppers für XML-Attribute

```
<stopper id="S6" spsId="6" previousId="" enteringSensor="B6" leavingSensor="B6.1"  />
<stopper id="S7" spsId="7" previousId="" enteringSensor="B7" leavingSensor="B5.1"  />
<stopper id="S8" spsId="8" previousId="" enteringSensor="B8" leavingSensor="B8.1" />
```

Abbildung 18: Beispieldefinition von Stoppern in der Beschreibungsdatei

Rfid-Geräte und Fixierstationen erhalten die zusätzliche Identifikation „spsId“, mit der sie angesteuert werden können. Eine Beispieldefinition von Weichen ist in der Abbildung 16 dargestellt.

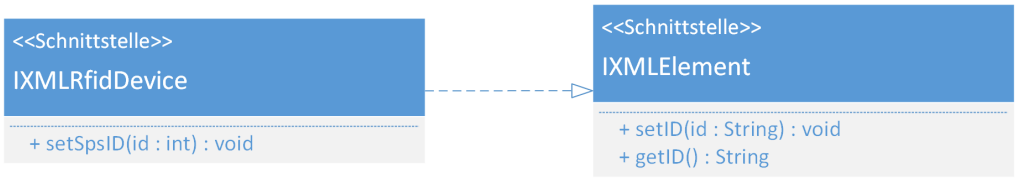

Abbildung 19: Übersicht über die Setter-Methoden eines Rfid-Gerätes für XML-Attribute

Nodes besitzen Attribute zum Definieren von jeweils einen Ein- und Ausfahrtssensor und einen Stopper. Optionale Attribute sind ein Rfid-Gerät und eine Fixierstation, da diese nicht an jedem Node vorhanden sein müssen, um Schlitten routen zu können. Unbedingt benötigt werden allerdings die Routeninformationen, wie andere Knoten erreicht werden können. Diese Routen sind als Kindelemente innerhalb des Nodes abgelegt und enthalten Informationen darüber, welche Knoten angefahren werden können („dest“), über welche Bahn dies geschieht („lane“) und welcher der nächste Knoten ist, an dem der Schlitten auf dieser Route eintrifft („nextHop“). Alle Angaben sind zwingend erforderlich, allerdings kann der Zielknoten („dest“) leer gelassen werden, diese Route wird dann als Standardroute verwendet. Außerdem sind Aktoren als Kindelemente abgespeichert. Weichen und Stopper können prinzipiell als Aktoren gekennzeichnet werden. Der genaue Aufbau der Setter ist in Abbildung 22 dargestellt.

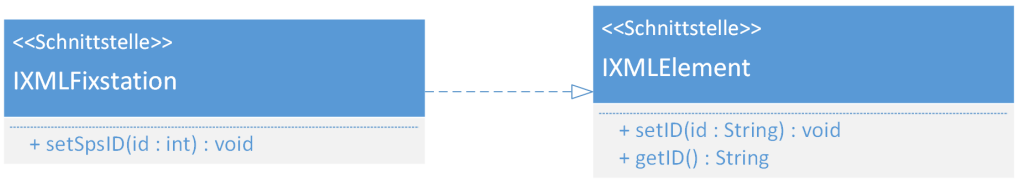

Abbildung 20: Übersicht über die Setter-Methoden einer Fixierstation für XML-Attribute

```
<device id="rfidStorage" spsId="1" />
<device id="rfidRobot" spsId="2" />
<device id="rfidEAStation" spsId="3" />
```

Abbildung 21: Beispieldefinition von Rfid-Geräten in der Beschreibungsdatei

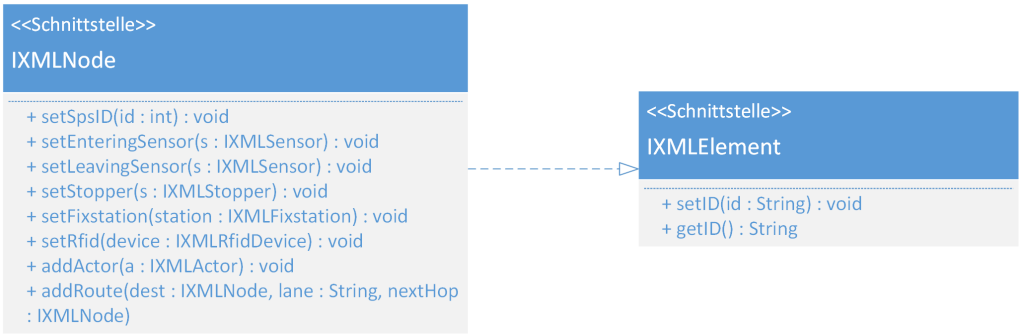

Abbildung 22: Übersicht über die Setter-Methoden eines Nodes für XML-Attribute

3.3 Auftragsmanagement

Das Auftragsmanagement bildet den Schlüsselpunkt zwischen der Kommunikationsschicht und der Abarbeitungslogik. Nach dem ein Auftrag im *ControlNetwork* eingegangen ist, wird dieser mittels der Methode *createOrder()* erzeugt. Als Parameter werden alle relevanten Informationen mitgegeben, um ein Auftragsobjekt zu erzeugen. Das erzeugte Objekt wird anschließend in eine Liste gespeichert, welche ebenfalls vor konkurrierenden Zugriffen geschützt wird. Dieser Aspekt muss gewährleistet werden, da im Programmablauf verschiedene Knoten versuchen werden gleichzeitig sich einen Auftrag zu beschaffen. Abschließend werden nach dem erzeugen alle Auftragslistener benachrichtigt, welche sich über das Interface *INewOrderListener* und der Methode *registerNewOrderListener()* registriert haben, benachrichtigt. Die registrierten Objekte werden in einer ArrayList verwaltet. Mittels der im Interface definierten Funktion *onNewOrder* wird das gerade erzeugte

Auftragsobjekt direkt an alle Interessenten geleitet (Siehe Abbildung 23).

Des weiteren besteht eine HashMap, welche die Zuordnung aller Knoten mit ihren zugehörigen alias Namen beinhaltet. Dazu muss sich zu beginn jedes Knotenobjekt mittels *addNodeMappingEntry()* bekannt machen. Das Key-Value-Paar wird für die Identifizierung des Auftrags benötigt, da dieser den Zielknoten als Objekt inne hält.

Die Beschaffung kann auf unterschiedliche weiße erfolgen. Für das anfordern eines Leerschlittens steht die Funktion *getOrderForEmptyCarriage()* zur Verfügung, welche die Liste nach einem Auftrag für einen Leerschlitten durchsucht und zurückgibt. Nach dem gleichen Prinzip existiert die Methode *getOrderForPalet()*, welche eine Paletten ID annimmt und nach dieser sucht.

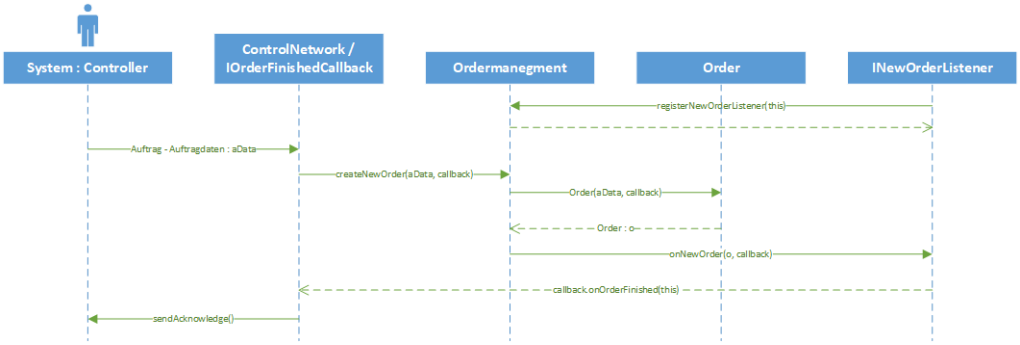

Abbildung 23: Sequenzdiagramm - Neuer Auftrag mit Bestätigung

3.4 Bahnmanagement

Das Bahnmanagement verwaltet die Positionen aller Schlitten, die sich aktuell auf den Bändern befinden. Hierzu steht für jeden Knoten auf dem Band eine eigene Liste bereit. Diese funktioniert wie eine Warteschlange, welche sich hinter dem Knoten einreiht. Passend dazu implementiert das Bahnmanagement das Interface *INodeQueue*, um auf die Listen der Knoten zuzugreifen. Des weiteren stehen die Funktionen *peek()* und *poll()* zum Auslesen und herausnehmen des jeweils ersten Warteschlangeneintrags, wie auch *addLast()* und *removeLast()* zum Hinzufügen oder entfernen eines Elements am Ende zur Verfügung. Des weiteren können Knoten mittels der Methode *checkNodeInList()* überprüfen, ob sich ein Schlitten in ihrer Liste befindet.

Eine weitere Funktionalität des Bahnmanagement ist die rückwärts suchende Leerschlittensuche. Welche zum tragen kommt, wenn ein leerer Schlitten angefordert wird. Hierfür werden ab dem Zielknoten rückwärts die ein-

zelnen Warteschlangen durchsucht, ob ein Leerschlitten gerade frei ist. Falls dies der Fall ist wird dieser genommen, anstelle eines neuen aus dem Leerschlittenlager.

3.5 Anlagenkomponenten

Die Logik, wie Schlitten behandelt werden, wenn sie auf der Bahn fahren, ist in den Kernkomponenten implementiert. Diese werden nach den Informationen in der Beschreibungsdatei vom Parser erstellt. Die einzelnen Instanzen werden dabei sich selbst überlassen und unterliegen keiner übergeordneten Instanz, die alles steuert. Dadurch, dass die Instanzen untereinander über Pointer beziehungsweise Events verbunden sind, entfällt einerseits der Verwaltungsaufwand zum Speichern aller Komponenten in Listen und andererseits die Suche der richtigen Instanz anhand einer Identifikation.

Sensoren Die Sensoren bilden die Basis für alle Aktionen, die durch eine Schlittenbewegung ausgelöst werden. Dafür benötigen die Sensoren eine Anbindung an eine Komponente, die Zugriff auf die Steuerung hat, von dem die Sensorwerte abgefragt werden können. Da auf einer Anlage möglicherweise sehr viele Sensoren sind wird nicht für jeden Sensor einzeln nachgefragt, sondern ein Sensor registriert sich auf die Änderung. Die Komponente, auf die ein Sensor zugreifen kann, muss das Interface *ISensorInterface* implementieren. Der Aufbau des Interfaces ist in Abbildung 24 dargestellt.

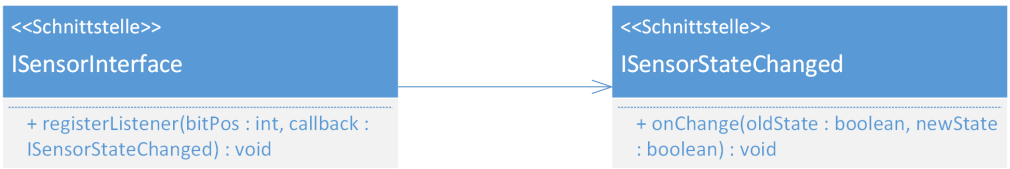

Abbildung 24: Mindestanforderungen an die Komponente zum Abfragen der Sensorwerte

Die Sensorwerte werden von der Steuerung als Byte-Array übertragen, in dem ein einzelnes Bit für einen Sensorwert steht. Ist das Bit an dieser Stelle 1 bedeutet das, dass sich ein Schlitten über den Sensor befindet, bei 0 befindet sich kein Schlitten über dem Sensor. Die Zuordnung, welches Bit welchen Sensor darstellt geschieht in der Beschreibungsdatei durch das Attribut „bitPos“. Die Position wird dabei von links kommend angegeben (siehe Abbildung 25). Der Sensor registriert sich bei seiner Instanz des

Abbildung 25: Veranschaulichung der Bit-Reihenfolge im Byte-Array der SPS beim Abfragen der Sensorwerte

ISensorInterface, sobald der Parser das Attribut aus der Beschreibungsdatei in das Objekt überträgt. Dazu ruft der Sensor *registerListener* auf und übergibt seine Bitposition und ein Callback, das als Lamda-Expression ausgedrückt wird. In dieser wird lediglich der gemeldete Wert auf eine steigende Flanke überprüft und im Sensor gespeichert. Ändert sich der Wert dabei werden alle Listener benachrichtigt, die sich auf diesen Sensor registriert haben. Die Funktionalität zum Registrieren ist im Interface *ISensor* vorgeschrieben (siehe Abbildung 26).

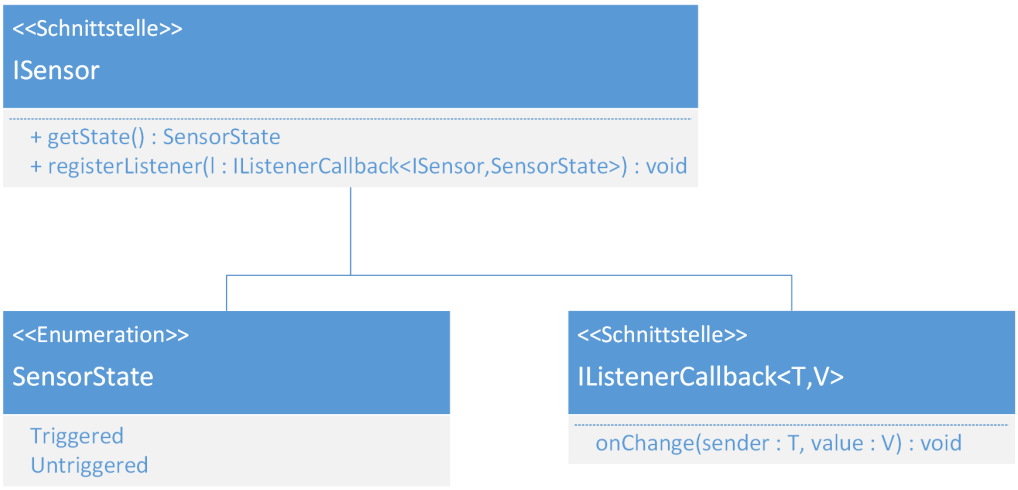

Abbildung 26: Aufbau und Abhängigkeiten des Interfaces ISensor

Stopper Die grundsätzliche Funktion von Stoppers besteht darin, Schlitten mit oder ohne Palette am Weiterfahren zu hindern. Sie sind ebenfalls an der Steuerung angeschlossen und können über diese mit Befehlen gesteuert werden. Jeder Stopper besitzt neben der lesbaren Identifikation, die auch auf dem jeweiligen Stopper angebracht ist, eine Nummer, mit der er von der Steuerung identifiziert werden kann. Diese wird über eine Setter-Methode vom Parser in die Stopper-Instanz übertragen und für die spätere Verwendung zwischengespeichert. Zudem werden vom Parser Ein- und Ausfahrtssensoren gesetzt. Wenn gültige Instanzen übergeben wurden registriert sich

der Stopper auf steigende Flanken an den Sensoren. Dies geschieht mit einer Lamda-Expression, in der dann abhängig ob Ein- oder Ausfahrtssensor entweder die Methode *onCartEntry* oder *onCartPassing* aufgerufen wird. Fallende Flanken werden nicht berücksichtigt, da für die Ausfahrt eines Schlittens extra Sensoren angebracht sind. Zudem würde die fallende Flanke die Ausfahrt eines Schlittens zu schnell anzeigen, sodass durch das Hochfahren eines Stoppers der Schlitten aus der Bahn geworfen werden könnte. Die Funktionalität eines Stoppers wird durch das Interface *IStopper* beschrieben (siehe Abbildung 27). Es definiert zwei Methoden, mit denen ein Stopper ein- oder ausgefahren werden kann. Damit sich ein Stopper steuern kann

Abbildung 27: Aufbau des Interfaces IStopper

Abbildung 28: Aufbau des Machineinterfaces für Stopper

benötigt er eine Komponente, die ihn an die Steuerung anbindet, an die der Stopper angeschlossen ist. Eine solche Komponente muss das Interface *IStopperMachine* (siehe Abbildung 28) implementieren, damit ein Stopper damit arbeiten kann. Es definiert eine Methode, mit der ein Stopper mit einer bestimmten Identifikation in eine bestimmte Position gebracht werden kann.

Die konkrete Implementierung ist in der Klasse *Sensor*. Sie implementiert dazu die Interfaces *IStopper* und damit auch *IXMLStopper*. Neben den durch das Interface vorgeschriebene Methoden definiert sie Hilfsmethoden, die bei der Ein- und Ausfahrt eines Schlittens ausgeführt werden. In diesen wird der Vorgänger aus- oder eingefahren, um eine Auffahrvermeidung möglich zu machen.

Fixierstationen Da Stopper Schlitten nur in Fahrtrichtung blockieren können aber nicht gegen Bewegungen entgegengesetzt der Fahrtrichtung oder zur Seite, sind an speziellen Stationen zusätzlich Fixierstationen angebracht. Diese bestehen aus einem Dreieckskeil, der an einem ausklappbaren Arm angebracht ist. Der Keil greift dabei in eine dafür vorgesehene Aussparung im Schlitten, wodurch dieser in alle Richtungen fixiert und durch die Keilform immer in die exakt gleichen Position gebracht wird. Besonders an Stationen, an denen andere Maschinen die Schlitten bestücken oder leeren ist das von großem Nutzen, beispielsweise an der Roboter- oder Lagerstation.

Eine Fixierstation erhält neben der Identifikation, die in der Beschreibungsdatei zur Referenzierung verwendet wird, eine Nummer, mit der sie von der Steuerung identifiziert wird. Diese wird über eine Setter-Methode vom Parser in die Fixierstation-Instanz übertragen und für die spätere Verwendung zwischengespeichert.

Die Funktionalität einer Fixierstation ist durch das Interface *IFixstation* definiert (siehe Abbildung 29). Es definiert zwei Methoden *fix()* und *release()* mit denen eine Fixierstation festgestellt oder gelöst werden kann.

Damit eine Fixierstation auf die Steuerung zugreifen kann, an der sie angeschlossen ist, benötigt sie eine Komponente, die das Interface *IFixstationInterface* implementiert. Dessen Aufbau ist in Abbildung 30 zu sehen. Mit der Methode *setFixstation* kann die Fixierstation mit einer bestimmten Identifikation aktiviert, also festgestellt, oder deaktiviert, also gelöst, werden.

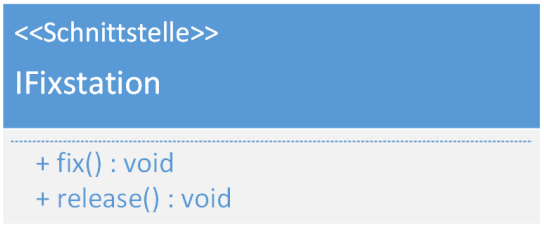

Abbildung 29: Aufbau des Interfaces IFixstation

Weichen Um verschiedene Bahnen miteinander zu verbinden und dazwischen Schlitten austauschen zu können, werden Weichen benötigt. Sie stehen zwischen zwei Bahnen und können Schlitten entweder auf das eine oder das andere Band einschleusen. Die Weiche kennt dabei nur die an sie angrenzenden Bahnen, sie ist daher der Routingebene 0 zuzuordnen.

Abbildung 30: Aufbau des Machineinterfaces für Fixierstationen

Sie erhält neben der Identifikation für die Referenzierung in der Beschreibungsdatei eine Nummer, mit der sie von der Steuerung, an der sie angeschlossen ist, identifiziert werden kann. Diese wird über eine Setter-Methode vom Parser in die Stopper-Instanz übertragen und für die spätere Verwendung zwischengespeichert. Zudem werden vom Parser beliebig viele Ein- und Ausfahrtssensoren gesetzt. Diese werden mit einem logischen Oder verknüpft und zur Koordination der Weiche verwendet. Die Oder-Verknüpfung wird durch die Registrierung der jeweils gleichen Methode für Ein- beziehungsweise Ausfahrtssensoren erreicht.

Zusätzlich werden noch beliebig viele Bahnen gespeichert, die an diese Weiche angeschlossen sind. Diese werden intern in einer Map gespeichert, bei der die Bahn der Key und der dazugehörige Wert, der an die Steuerung geschickt werden muss damit diese die Weiche richtig stellt, der Value ist.

Die Komponente, die die Anbindung an die Steuerung übernimmt, muss das Interface *ISwitchInterface* (siehe Abbildung 32) implementieren. Dieses schreibt eine Methode *setPosition* vor, mit der eine Weiche mit einer bestimmten Identifikation in eine bestimmte Position gebracht werden kann, die durch eine Zahl angegeben wird.

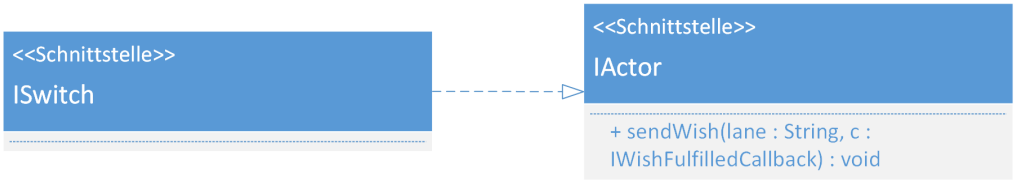

Abbildung 31: Aufbau des Interfaces ISwitch

Rfid-Geräte Rfid[1]-Geräte werden auf der Anlage benötigt, um Paletten eindeutig zu identifizieren. Diese besitzen an der Seite einen Rfid-Chip, auf dem eine Zahl zwischen 0 und 255 gespeichert werden kann. Diese ist von

[1]Radio-Frequency Identification, Funktechnik für extrem kurze Distanzen

Abbildung 32: Aufbau des Interfaces ISwitchInterface

Beginn an auf dem Chip gespeichert und wird vom Transportmanager nicht überschrieben. Dieser liest lediglich die Id aus, um zu bestimmen, zu welchem Knoten die Palette muss.

Die einzelnen Rfid-Geräte werden in der Beschreibungsdatei definiert. Sie erhalten die obligatorische Id, die für die Referenzierung verwendet wird. Zusätzlich erhalten sie noch eine Id, mit der das Gerät von der Steuerung identifiziert wird.

Die Funktionalität eines Rfid-Geräts wird vom Interface *IRfidDevice* vorgegeben, welches in Abbildung 33 dargestellt ist.

Die Verbindung zur Steuerung übernimmt eine separate Komponente, die das Interface *IRfidInterface* implementieren muss. Der Aufbau dessen ist in Abbildung 34 dargestellt.

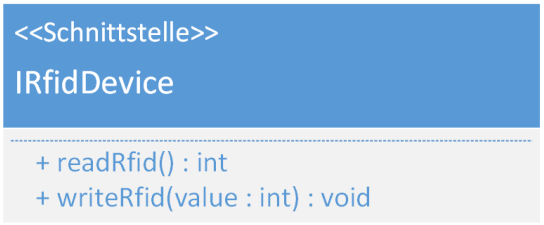

Abbildung 33: Aufbau des Interfaces IRfidDevice

Abbildung 34: Aufbau des Interfaces IRfidInterface

Knoten Knoten stellen auf der Anlage logische Punkte dar, zu denen theoretisch Schlitten transportiert werden können. Sie übernehmen die Weiter-

leitung von Schlitten und die Abarbeitung von Aufträgen. Sie haben dazu Zugriff auf folgende Verwaltungskomponenten:

- Auftragsverwaltung
- Queueverwaltung

Außerdem besitzen Knoten jeweils einen Ein- und Ausfahrtssensor, einen Stopper und ein optionales Rfid-Gerät.

Mithilfe der Queueverwaltung kann ein Knoten auch ohne angeschlossenes Rfid-Gerät ermitteln, ob es sich bei dem einfahrendem Schlitten um einen beladenen oder leeren Schlitten handelt. Eine Garantie hat der Knoten allerdings nicht, da er die Information nicht überprüfen kann. Daher sollten möglichst alle Knoten mit einem Rfid-Gerät ausgestattet sein. Bei der Einfahrt eines Schlittens wird immer das erste Element in der Queue betrachtet. Ist in der Queue kein Schlitten vorhanden aber physisch einer eingefahren, wird ein neuer Schlitten an den Anfang eingefügt. Bei der Ausfahrt eines Schlittens wird dieser aus der eigenen Queue entfernt und in die Queue des als NextHop eingetragenen Knotens eingefügt.

Damit ein Knoten das Ziel eines Schlittens oder einer Palette abfragen kann, hat er Zugriff auf die Auftragsverwaltung. Diese wird durch ein Objekt des Typs *IOrderManagement* repräsentiert. Die Abfrage findet bei der Einfahrt eines Schlittens statt. Diese wird ausgelöst, wenn ein Schlitten über den Einfahrtssensor fährt. Zuerst wird geprüft, ob es sich bei dem Schlitten um einen beladenen oder leeren handelt. Dies kann entweder über das Rfid-Gerät oder über die Queueverwaltung geschehen. Wenn es sich um einen Schlitten mit Palette handelt wird anhand der Palettennummer der Transportauftrag und damit das Ziel ermittelt. Handelt es sich um einen leeren Schlitten oder ist kein Rfid-Gerät hinterlegt wird zuerst auf einen bereits gespeicherten Transportauftrag im Schlitten geprüft. Ist dieser gesetzt wird dieser abgearbeitet, andernfalls wird ein Auftrag für einen leeren Schlitten von der Auftragsverwaltung geholt und im Schlitten gespeichert.

Die Abarbeitung eines Schlittens folgt immer einem festen Muster: Es wird geprüft, ob auf dem abzuarbeitenden Schlitten ein Auftrag gesetzt ist, wenn nicht wird die Abarbeitung sofort beendet. Wenn doch prüft der Knoten, ob es sich beim Zielknoten um sich selbst handelt, wenn ja wird der Auftrag durch den Aufruf der Methode *finish* abgeschlossen und die Abarbeitung beendet. Andernfalls werden die Routeninformationen zum Zielknoten gelesen, die Aktoren gesteuert und der Schlitten dann weitergeleitet.

4 Problemstellungen und Lösungen

4.1 Auffahrvermeidung

Um zu verhindern, dass Schlitten auf dem Haupt- oder EA-Band aneinanderstoßen, wurde eine Logik realisiert, die dies verhindert. Dazu wurde keine extra Logik in der Software implementiert sondern Einstellungen in der XML-Beschreibungsdatei ausgenutzt, in der Einfahrtssensoren von Stoppern absichtlich „falsch“ gesetzt wurden. Der Einfahrtssensors eines Stoppers wird dabei auf den Ausfahrtssensor seines Vorgängers gesetzt wenn der Vorgänger nicht Teil eines Nodes ist.

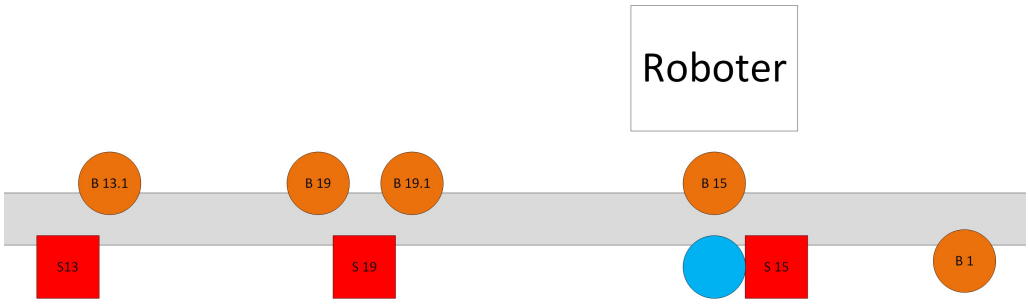

Abbildung 35: Ausschnitt eines Bahnabschnitts zur Veranschaulichung der Auffahrlogik. Orange Kreise sind dabei Sensoren, rote Quadrate Stopper und blaue Kreise Rfid-Geräte

In Abbildung 35 ist der Bahnabschnitt zwischen Lager und Roboter abgebildet. Als Einfahrtssensor für S15 wurde B19.1 anstatt B15 verwendet. Der Einfahrtssensor von S19 ist B13.1. Als Ausfahrtssensoren wurden jeweils die Sensoren nach den Stoppern eingetragen.

Damit ein Stopper seinen Vorgänger steuern kann muss dieser ebenfalls in der Datei hinterlegt sein. Dabei ist S19 bei S15 und 13 bei S15 hinterlegt. Das hat zur Folge, dass wenn ein Schlitten einen Stopper verlässt, also über den Ausfahrtssensor fährt, er zeitgleich auch in den nächsten Stopper einfährt und dadurch der vorhergehende wieder Stopper ausgefahren wird. Es kann zu Komplikationen kommen, wenn ein Stopper sowohl ein Vorgänger als auch Teil eines Nodes ist. Schlitten könnten dann weitergefahren werden, obwohl der Node seine Arbeit noch nicht abgeschlossen hat.

4.2 Kurvensteuerung

Die Kurvensteuerung wurde ebenfalls aus der SPS herausgenommen und im Transportmanager implementiert, da so der Betreuer der Anlage nur ein System warten muss. Die Logik funktioniert wie die in Abschnitt 4.1 beschriebene Auffahrvermeidung.

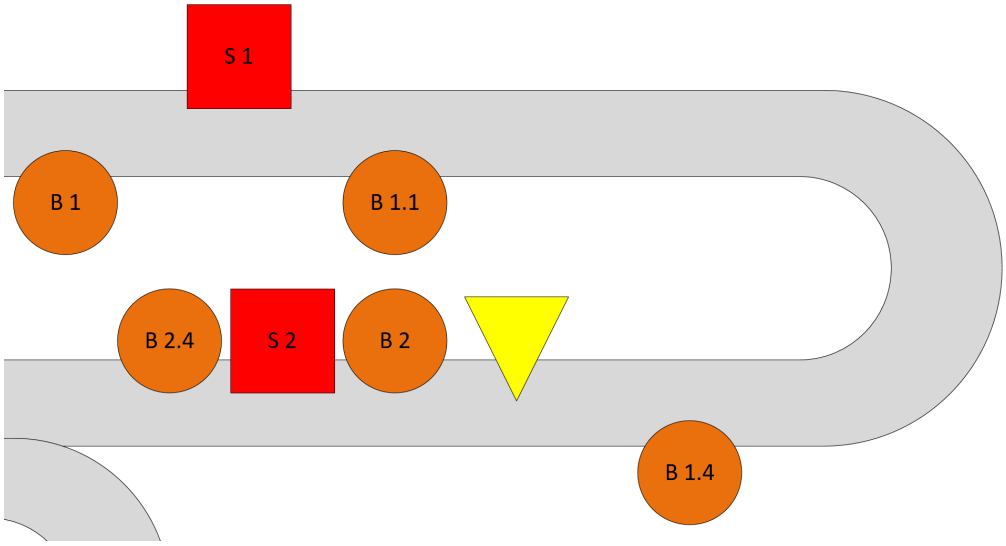

Abbildung 36: Ausschnitt eines Bahnabschnitts zur Veranschaulichung der Kurvensteuerung. Orange Kreise sind dabei Sensoren und rote Quadrate Stopper

In Abbildung 36 ist beispielhaft eine Kurve aus der Anlage abgebildet. Die Einfahrtssensoren sind hier ebenfalls „falsch“ gesetzt, damit der Vorgänger sofort hochgefahren wird, sobald ein Schlitten diesen verlässt. Dazu ist als Einfahrtssensor für den Stopper S2 der Sensor B1.1 eingetragen. Die Ausfahrtssensoren sind jeweils die direkt nach dem Stopper.

4.3 Konkurrierender Zugriff auf Aktoren

Durch das eventbasierte Design können mehrere Instanzen gleichzeitig arbeiten und die Daten der Sensoren lesen. Weichen stellen jedoch eine Besonderheit dar, da diese von zwei, prinzipiell aber von n Nodes gesteuert werden. Deswegen können Aktoren, zu denen Weichen zählen, nicht direkt gesteuert werden sondern nehmen Wünsche entgegen, die andere Instanzen, hier also Nodes senden können. Diese werden intern in einer Liste gespeichert und nach einer bestimmten Reihenfolge abgearbeitet. In der jetzigen Version ist keine gesonderte Logik zur Optimierung der Weichen imple-

mentiert sondern Wünsche werden in der Reihenfolge abgearbeitet wie sie gesendet wurden. Dies kann aber durch Austausch der Implementierung einfach geändert werden.

Um die Instanz zu benachrichtigen, dass ihr Wunsch erfüllt wurde, kann ein Callback angegeben werden, das von der Weiche aufgerufen wird, wenn sie sich in die gewünschte Stellung gebracht hat. Der Node fährt durch das Callback seinen Stopper ein, um den Schlitten zu seinem Ziel zu bringen. Die Weiche erkennt die Einfahrt und Ausfahrt des Schlittens durch die eingetragenen Sensoren. Bei der Ausfahrt wird die Weiche benachrichtigt, die intern den Wunsch abschließt und damit das Callback aufruft. Sollten weitere Wünsche vorliegen werden diese abgearbeitet, andernfalls endet hier die Abarbeitung. Die Zugriffsmethoden auf das Wunschsystem müssen vor konkurrierenden Zugriffen geschützt werden, daher sind diese mit synchronized deklariert.

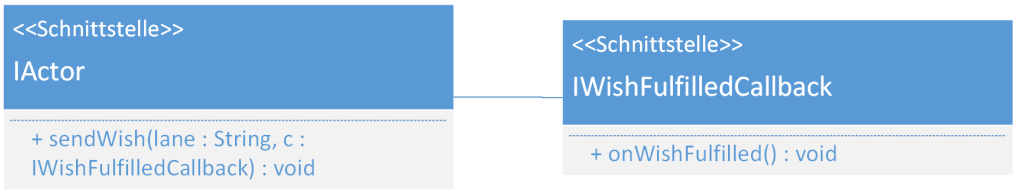

Abbildung 37: Darstellung der Schnittstelle eines Aktors

Um Wünsche entgegennehmen zu können, muss das Interface IActor (siehe Abbildung 37) implementiert werden. Die Wünsche basieren auf der Routingebene 0 (siehe Absatz 4.4).

4.4 Routing

Ziel des Transportmanagers ist es, Schlitten an die gewünschte Position zu bringen. Da die Anlage nicht fest in der Software hinterlegt sondern in einer externen Datei beschrieben ist, müssen auch die Routinginformationen in dieser hinterlegt sein.

```
<node id="node1" ...>
    <routingInformations>
        <route destination="node3" lane="exampleLane" nextHop="node2" />
        <route destination="" lane="otherLane" nextHop="node4" />
    </routingInformations>
</node>
```

Abbildung 38: Darstellung der Definition von Routen in einem Node

Ein Beispiel für Routendefinitonen ist in Abbildung 38 zu sehen. Dort werden für den Node „node1“ zwei Routen definiert, wovon eine nur für Schlitten gilt, die als Ziel „node3“ haben. Eine Route mit einem leeren Zielknoten sind Standardrouten, die angewendet werden, wenn keine Route vorher zutrifft. Pro Node darf es nur eine Standardroute geben.

Das Attribut „lane“ wird für das Routing auf Bahnebene (Routingebene 0) benötigt. Auf dieser Ebene arbeiten Aktoren. Routingebene 1 beschäftigt sich mit dem Routing über Nodes hinweg, wofür das Attribut „nextHop“ benötigt wird. Mithilfe dessen können Schlitteninformationen weitergegeben werden, was eine Behandlung von Schlitten auch ohne angeschlossenen Rfid-Leser ermöglicht. Allerdings darf in Bahnsegmente, die einen Node ohne Rfid-Leser als Ende haben, kein Schlitten eingesetzt oder entfernt werden.

Die Routinginformationen müssen immer vollständig sein, d. h. es muss von jedem Node aus eine Route zu jedem anderen Node vorhanden sein. Ist dies nicht der Fall, könnten Schlitten nicht verarbeitet werden und es kommt zur Blockade.

5 Use-Cases

5.1 Start der Anwendung

Nach dem Starten der Anwendung wird gefragt, ob, falls vorhanden, noch offene Aufträge bzw. Nachrichten wiederhergestellt werden sollen. Ist dies der Fall wird die Pestenzrsistenzroutine gestartet, welche in Kapitel 3.1.1 genauer beschrieben ist. Nachdem der letzte Zustand wiederhergestellt wurde, wird wie nach einem normalen Programmstart fortgefahren.

Zu beginn werden die Übergabeargumente eingelesen und validiert. Danach wird der ThreadPool erzeugt, da dieser von den anderen Komponenten verwendet wird. Anschließend werden die Verwaltungskomponenten Auftragsmanagement und Bahnmanagement erzeugt, damit diese später beim erzeugen der Anlagenkomponenten zur Verfügung stehen. Dazu werden sie der ModelFactory übergeben, welche dann für das parsen der Beschreibungsdatei verwendet wird. Als nächstes werden die Anlagenkomponenten gemäß der Definition in der Beschreibungsdatei erzeugt und die Verbindungen zu den angrenzenden Systemen aufgebaut.

5.2 Neuer Transportauftrag

Nach dem Versenden eines neuen Transportauftrags, wird dieser wie in Kapitel 3.1.1 beschrieben, entgegengenommen und verarbeitet. Nach dem erfolgreichen Anlegen des Auftrags wird eine erste Bestätigung zurückgesendet. Diese teilt dem Sender mit, dass sein Auftrag erfolgreich entgegengenommen wurde und so bald wie möglich abgearbeitet wird.

Das Auftragsmanagement, welches alle Aufträge verwaltet, benachrichtigt alle registrierten Knoten, dass ein neuer Auftrag eingetroffen ist. Danach wird jeder dieser Knoten überprüfen, ob der neue Auftrag für ihn bestimmt ist. Dafür prüft jeder Knoten, ob sich momentan ein Schlitten bei ihm steht. Falls dies der Fall ist, wird noch überprüft, ob der Schlitten bereits bereits für einen anderen Auftrag verwendet wird. Ist dies der Fall, kann der gestellte Transportauftrag nicht bearbeitet werden, da dieser noch nicht abgearbeitet wurde. Dies garantiert, dass sich keine Aufträge überschneiden. Bedeutet, dass ein Auftrag für einen Schlitten nur ausgeführt wird, wenn diese aktuell keinen offenen Auftrag besitzt.

Falls zum Auftrag der passende Schlitten verfügbar ist, wird nachgesehen, um was für eine Art von Auftrag es sich handelt (Siehe Kapitel 3.1). Anschließend wird geprüft, ob die Palette zum Auftrag passt und dieser auf den Schlitten gespeichert, damit das Routing an Knoten ohne RFID-Leser funktioniert.

5.3 Bewegungen auf der Bahn

Die Bahn besteht aus insgesamt sechs Knoten, wobei drei davon mittels Kommandos angefahren werden können und einer als Ziel für das aufräumen der nicht mehr benötigten Schlitten verwendet wird. Ob ein Knoten als Ziel verwendet werden kann, hängt davon ab, ob in der Beschreibungsdatei ein Alias für diesen vergeben wurde (Siehe Kapitel 3.3).

Fährt ein Schlitten an einen Knoten ein, wird als erstes auf Konsistenz des Bahnmanagements überprüft. Hier greift das in Absatz 5.4 beschriebene Fehlerhandling. Ist der Zustand konsistent, so wird das Ziel aus dem gespeicherten Auftragsobjekt des ankommenden Schlittenobjekts ausgelesen und vom Knoten verglichen, ob er der gespeicherte Zielknoten ist. Ist dies nicht der Fall, greift auf das in Kapitel 4.4 beschriebene Routing, welches den Schlitten automatisch an den nächsten verantwortlichen Knoten weiterleitet.

Falls der Zielknoten erreicht wurde, so wird der im Auftrag hinterlegte Callback aufgerufen, welcher für das Abschließen des Auftrags verantwortlich ist. Genaue Abarbeitungslogik kann dem Sequenzdiagramm in Abbildung 23 Zu guter Letzt wird der Auftrag aus der Liste des Auftragsmanagements entfernt und die zweite Bestätigungsnachricht an den Sender des Auftrags zurückgesendet.

5.4 Fehlerhandling

Das Fehlerhandling wird praktischerweise durch die generisch strukturierte Softwarearchitektur behandelt. Aufgrund dessen, dass keine harten Strukturen existieren, sondern alles symbolisch über die Definitionen in der Konfigurationsdatei miteinander verbunden wird, regeneriert sich das System nach Fehlerzuständen automatisch. Beispielsweise ist es kein Problem vor Programmstart oder während des Betriebs Schlitten mit Paletten in das System einzuschleusen oder zu entnehmen. Im Verlauf des Betriebs werden nicht registrierte Paletten in die Programmlogik mit aufgenommen und können anhand ihrer ID normal angesprochen werden. Die Inkonsistenz wird erkannt, sobald eine nicht registrierte Palette an einem Knoten mit RFID-Leser einfährt. Hierbei wird immer überprüft, ob es sich um eine erwartete Palette handelt. Ist dies nicht der Fall wird diese in das laufende System integriert und kann anschließend voll verwendet werden. Ebenfalls sind leere Schlitten auf den Bändern kein Problem. Wurde einem Schlitten kein Ziel zugeordnet oder hat er dieses verloren wird automatisch eine Rückleitungsroutine gestartet. Diese räumt den ziellosen und eventuell blockierenden Schlitten automatisch auf. Dafür wird ihm als Ziel das Leerschlittenband zugeordnet. Diese Routine greift auch, falls sich bei Programmstart noch leere Schlitten auf dem Band befinden oder angeforderte Schlitten nicht mehr benötigt werden.

5.5 Herunterfahren des Systems

Wie in Kapitel 3.1.1 beschrieben, nimmt das Modul den SShutdownBefehl entgegen und wertet diesen aus. Nach Erhalt des Befehls werden alle registrierten ShutdownListener benachrichtigt, welche dann den Prozess des herunterfahrens ausführen. Hierfür wird der ThreadPool beendet, welcher alle laufenden Threads benachrichtigt. Dies geschieht mittels eines beim erzeugen des Threads übergeben Flags, welches zum beenden gesetzt wird.

Alle Jobs können nun ihren Zyklus noch abschließen und beenden sich danach selbst. Wenn alle Jobs ihre Routine abgeschlossen haben, beendet sich das Programm automatisch.